Predictive Runtime Analysis of Multithreaded Programs*

Feng Chen and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana - Champaign, USA
{fengchen,grosu}@uiuc.edu

Abstract. We present a technique to predict property violations in multi-threaded programs from successful executions. An appealing aspect of our technique is that it is entirely automatic; another is that no special simulation or modeling infrastructure is needed. All the user needs to do is to provide the multi-threaded system and the property to check. An observer is automatically generated from the property and an instrumentation procedure based on vector clocks automatically modifies the program to emit relevant events to the observer. By making intensive use of a dynamically computed generalized "happens-before" causal partial order that is refined with control-flow and data-flow dependency information obtained apriory via static analysis of the program, the observer is able to build from one concrete execution trace a set of abstract execution traces. Those abstract execution traces correspond to concrete executions that have not necessarily happened but "were close to happen" and could indeed happen in another execution of the system under a different thread scheduling. The predictive aspect of our technique comes from the fact that some of those executions may violate the property. If that is the case, a counter-example is provided. The technique has been implemented in the context of Java and has been shown to be useful via a series of experiments.

1 Introduction

An execution of a multi-threaded program is the result of a certain input and a certain schedule of threads. Due to the nondeterministic nature of thread scheduling, the program may very likely follow a different execution when it runs again, even with the same input. This inherent characteristic of multi-threaded system makes them difficult to test and debug. This paper introduces a technique to correctly detect concurrency errors from observing just one execution of the program, via a totally automatic instrumentation of the program to emit "more than the obvious" information to an external observer. The particular execution that is observed need *not* hit the error; yet, errors can be predicted in other executions without false alarms. The observer, which can potentially run on a different machine, will never need to see the code which generated those events but still be able to correctly predict errors that could really appear in other executions, together with a scenario (counterexample) under which the error would appear.

There are several other approaches in the literature aiming at detecting potential errors in concurrent system by examining particular executions. Some of these aim at verifying general purpose behavioral properties, including temporal ones, [17, 16, 15] and are inspired from efforts in debugging distributed systems based on Lamport's "happens-before" causal partial ordering on runtime events [10]. Others aim at dynamic behavior reduction and have been designed to work best for particular properties of interest, such as data-race and/or atomicity detection by means of lock-set algorithms [14, 7]. These previous efforts focus on either soundness or completeness: approaches based on the "happen-before" relation are sound but have limited coverage over interleavings, thus resulting in more false positives (missing errors); lock-set based approaches produce fewer false positives but suffer from false negatives (false alarms). There are also works combining "happen-before" and lock-set techniques, e.g., [12], aiming at achieving a better balance. At our knowledge, the approach presented in this paper is the first to combine dynamic analysis, based on a special "happen-before" causal partial order, with control-flow and data-flow dependency static information of the multi-threaded program, resulting in a practical technique for precise violation prediction of general-purpose properties, with significantly less coverage compromise than the other "happen-before" approaches.

The presented predictive analysis technique lays somewhere between testing and model checking. Testing because one essentially runs the system and observes its runtime behavior in order to detect errors, and model checking because the causal partial order can be regarded as an abstract model of the program which

* Supported by the joint NSF/NASA grant CCR 0234524, NSF CCF-0448501, and NSF CNS-0509321.

can further be investigated exhaustively by the observer. To avoid false alarms, the permutations of abstract events analyzed by the observer should be consistent with the semantics of the original program. Previous approaches based on the "happen-before" idea extract the causal order based on just the dynamic inter-thread communication. This is a simple way to obtain a causal order that guarantees the correctness of all its linearization. But since *all* interactions among threads are considered, the obtained causal partial orders are rather restrictive, allowing a reduced number of linearizations and thus of errors that can be detected. By considering information about the static structure of the multi-threaded program in the computation of the causal partial order, we can filter out irrelevant thread interactions and thus obtain a more relaxed causality, allowing many more valid permutations. Lock-set related approaches focus on synchronization blocks of events, which are easy to handle but unsound. We also take synchronization into account in our approach, in the sense that events protected by locks can only be permuted in blocks. This way, our approach borrows comprehensiveness from lock-set approaches without giving-up soundness.

We implemented our approach in Java, based on code instrumentation and offline analysis. The multi-threaded program to test is instrumented to generate detailed execution traces and save them into log files, which are filtered and then analyzed to predict violations of desired properties. The prototype has been applied on several non-trivial applications and the results seem promising.

2 Dynamic/Static Causal Dependence

Previous efforts in predictive runtime analysis [17, 16, 15] were based on "happen-before" causalities [10] extracted dynamically from running multi-threaded systems by considering *all causal dependences* on all the events. The obtained causal partial orders contained all the information needed by a hypothetical observer to construct various other sequences of *concrete* program states, sequences that could happen under different thread schedulings. Online observers then abstracted, or projected, these sequences of states by removing irrelevant information and then checking them against the system requirements. The predictive aspect of those techniques was due to the fact that some of those executions could violate the requirements, despite the fact that the actual execution that took place was successful. The major point of this paper is that one can significantly relax the previous dynamic causal partial orders by dropping some previously considered causal dependencies, based on *control-flow and data-flow information* obtained by *static analysis*. Observers can then generate sequences of states that are *concrete only up-to-relevant events*, this way being able to explore many more thread interleavings, thus increasing their predictive power.

Let us consider the program in Figure 1 (A). Suppose that the property φ to check is "$e > 0$ *implies in the past* $c > 0$". This program violates φ, because it admits some "bad" executions, e.g., $(L_{11}L_{21}L_{12}L_{22}L_{23}L_{13}L_{24})$. Normal testing may *not* discover the bug, since there are many executions not violating φ, e.g., $(L_{11}L_{21}L_{12}L_{13}L_{22}L_{23}L_{24})$ depicted in Figure 1 (B);

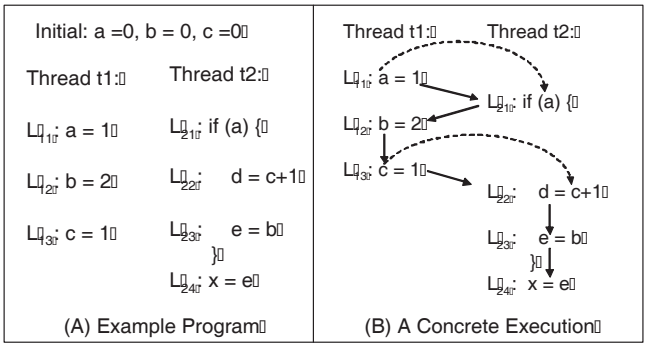

Fig. 1. Example for causal orders

solid arrows show the execution order and dotted arrows show the *causal partial order* on the reads and writes of shared variables: reads of a and c in L_{21} and L_{22}, respectively, causally depend upon the writes in L_{11} and L_{13}. It is worth mentioning that the causal partial order is actually between events; we use statement labels here only to keep the discussion brief, but actually a statement may produce multiple events. Refined discussion is given shortly. An observer of an execution trace cannot consider permutations of events violating the causal partial order, because it may construct invalid executions. For example, if L_{22} is permuted before L_{13}, the value written to d at L_{22} is not the observed 2 anymore. To build a correct execution the observer would need to re-execute the code, a heavy task that we deliberately avoid in this paper; the challenge is for the observer to build other valid executions *using only the information that it already has* from observing one execution trace.

Consider the partial order obtained by combining this causality with a *total* ordering on events generated by each thread, as in [17, 16, 15, 10]. An observer can then explore only two other consistent linearizations, namely $(L_{11}L_{12}L_{21}L_{13}L_{22}L_{23}L_{24})$ and $(L_{11}L_{12}L_{13}L_{21}L_{22}L_{23}L_{24})$, neither of them violating φ (because L_{13}

occurs before L_{23} in both). With the static/dynamic approach to causality proposed in this paper, observers are able to explore more valid executions, in particular some executions in which L_{13} and L_{23} can be permuted. The key idea is to build upon the fact that the value written to e at L_{23} does not depend on the value of d at L_{22}, so implicitly it does not depend upon the value written to c at L_{13}. Therefore, the observer can afford to ignore the causal dependency of L_{22} upon L_{13}. Even though some of the corresponding permutations, such as $(L_{11}L_{21}L_{12}L_{22}L_{23}L_{24}L_{13})$, may seem not to be observationally consistent with the original execution, in this case because the value written to d at L_{22} would be different from the one observed, they are still valid w.r.t. the relevant events, namely, the write of c at L_{13} and that of e are L_{23}. That means that there is indeed some execution of the multi-threaded program that generates precisely the predicted sequence of relevant events. What makes the new technique effective is the fact that some of the traces built by relaxing the causal dependency using static information, e.g., $(L_{11}L_{21}L_{12}L_{22}L_{23}L_{24}L_{13})$, can violate the requirements.

To make our point clearer, consider now the property "$e > 0$ *implies in the past* $b > 0$". Then L_{23} cannot be permuted before L_{12} since such permutation may change the value of e in L_{23}, leading to an invalid execution. The situation is different when the desired property concerns only occurrences of events, for example, "*a write on e implies a write on b in the past*". Since values are not important anymore, the data-flow dependence from L_{12} to L_{23} can be dropped: no matter whether L_{12} has been executed or not, the write on e at L_{23} will happen anyway. So $(L_{11}L_{21}L_{22}L_{23}L_{24}L_{12}L_{13})$ is a possible execution w.r.t to this property; moreover, it violates the property. However, note that L_{23} cannot be permuted before L_{11}: L_{23} has a control-flow dependence upon L_{21} and the latter cannot be permuted before L_{11} because of an obvious data-flow dependence justifying the existence of the write to e in L_{23}. Note that violations of "*a write on e implies a write on a in the past*" cannot be predicted from the observed trace.

The point we are trying to make here is that one can significantly improve over existing work in predictive runtime analysis if one considers structural information about the program, obtained via static analysis of its source code. Observers of multi-threaded runs can generate other potential valid runs that can yield sequences of states that are complete only up-to-relevant events. This way observers can explore more potential interleavings, increasing their predictive power. In the remaining of this section we formalize the notions of causal dependency and consistent runs discussed above.

2.1 Events and Traces

Definition 1. *An* ***event*** *is a mapping of* ***attributes*** *into corresponding* ***values****. Let* Events *be the set of all events. A* ***trace*** *is a finite sequence of events generated by an execution of a (multi-threaded) program. From now on, we assume an arbitrary but fixed trace τ, let ξ denote the set of all events in τ, and let $<_\tau$ denote the obvious total order on ξ: $e <_\tau e'$ iff e occurs before e' in τ.*

Figure 2 shows the trace of concrete events for the execution in Figure 1 (B). We call these events "concrete" because later, depending on the property to check, we will automatically filter the trace by removing irrelevant events and/or irrelevant attributes of events. Events in Figure 2 contain some typical attributes, including "thread", "statement", "type", "target", and "state". The thread and the statement say where the event was generated; e.g., e_1 was generated in t_1 at statement L_{11}. Every event has a certain type. There are two types of events in Figure 2, *read* and *write* of variables. There can be other event types, such as *begin* and *end* of function calls.

e_1 : $(thread = t_1,\ stmt = L_{11},\ type = write,\ target = a,\ state = 1)$
e_2 : $(thread = t_2,\ stmt = L_{21},\ type = read,\ target = a,\ state = 1)$
e_3 : $(thread = t_1,\ stmt = L_{12},\ type = write,\ target = b,\ state = 2)$
e_4 : $(thread = t_1,\ stmt = L_{13},\ type = write,\ target = c,\ state = 1)$
e_5 : $(thread = t_2,\ stmt = L_{22},\ type = read,\ target = c,\ state = 1)$
e_6 : $(thread = t_2,\ stmt = L_{22},\ type = write,\ target = d,\ state = 2)$
e_7 : $(thread = t_2,\ stmt = L_{23},\ type = read,\ target = b,\ state = 2)$
e_8 : $(thread = t_2,\ stmt = L_{23},\ type = write,\ target = e,\ state = 2)$
e_9 : $(thread = t_2,\ stmt = L_{24},\ type = read,\ target = e,\ state = 2)$
e_{10} : $(thread = t_2,\ stmt = L_{24},\ type = write,\ target = x,\ state = 2)$

Fig. 2. Trace for the execution in Figure 1

The meaning of the target and state of an event is related to the event type. For the variable access (read or write) type, e.g., the target is the name to the accessed variable and the state contains the value which is read from or written to the variable. For example, e_1 is a write of value 1 to variable a. (The situation is more complicated in the context of object field or array element access, because of the concrete address of the object/array and the index of the element; the state of the access event is slightly more complex in our implementation, but, for simplicity, we here only discuss primitive variable accesses.) For function related events, the target is the signature of the function and the state contains the input arguments (for *begin* events) or the return value (for *end* events). One can easily include more information into an event by adding new attribute-value pairs. We

use *attribute*(e) to refer to the value of *attribute* of event e. For example, in Figure 2, *thread*(e_1) = t_1 and *target*(e_4) = c. Events have unique names: even though two events contain the same value for every attribute, they are still assumed different.

When the trace τ is checked against a property φ, most likely not all the attributes of the events in ξ are needed; some events may not even be needed at all. For example, to check data races on a variable x, the states of the events of type *write* and *read* on x are not important; also, updates of other variables or function call events are not needed at all. We next assume a generic *filtering function* that can be instantiated, usually automatically, to concrete filters depending upon the property under consideration:

Definition 2. *Let α_φ: $\xi \to$ Events be a partial function, called a* ***filtering function****. The events on which α_φ is defined are called* ***concrete relevant events****. The image of α_φ, that is $\alpha_\varphi(\xi)$, is written more compactly ξ_φ; its elements are called* ***abstract relevant events****, or simply just* ***relevant events****.*

This abstraction will play a crucial role in increasing the predictive power of our analysis approach. That is because, in contrast to ξ, the more abstract ξ_φ will allow many more valid permutations of abstract events: instead of calculating permutations of ξ and then abstracting them into permutations of ξ_φ like in [17, 16, 15], we will calculate *directly* valid permutations of ξ_φ. Our goal is therefore to compute the precise causal partial order on abstract events in ξ_φ by analyzing the dependence among concrete events in ξ.

2.2 Control-Flow and Data-Flow Dependence

Without additional information about the structure of the program that generated the event trace τ, the least restrictive causal partial order that an observer can extract from τ is the one which is total on the events generated by each thread and in which each write event of a shared variable precedes all the corresponding subsequent read events. This is investigated and discussed in detail in [16]. In this section we show that one can do much better than that if one uses appropriately control-flow and data-flow dependence information that can be obtained via static analysis of the original program.

Intuitively, event e *depends upon* event e' in τ, written $e' \sqsubset e$, iff a change of e' may change or eliminate e. This tells the observer that e' *should occur before e in any consistent permutation of τ*. We discuss two kinds of dependence: (1) *control-flow dependence*, written $e' \sqsubset_{ctrl} e$, when a change of the state of e' may eliminate e (e.g., $e_2 \sqsubset_{ctrl} e_5$ in Figure 2); (2) *data-flow dependence*, written $e' \sqsubset_{data} e$, when a change of the state of e' may lead to a change in the state of e (e.g., $e_5 \sqsubset_{data} e_6$ in Figure 2). Our dependence relation $\sqsubset$ will merge the control-flow and the data-flow dependences (it will be the transitive closure of their union). For the trace in Figure 2, one can infer $e_1 \sqsubset e_6$: $e_1 \sqsubset_{data} e_2 \sqsubset_{ctrl} e_5 \sqsubset_{data} e_6$. In this subsection we formalize the two types of dependence. Even though their intuitions and motivations are relatively straightforward, the technical details are quite intricate.

One may notice that some concepts discussed below are similar to the work on program slicing [18], which tries to compute the parts of a program that may affect values computed at some specific points. But since our approach aims at a different goal, namely generating sound permutations of property-relevant events, we encounter different challenges here, resulting in different solutions. First, the relevant events are detected at runtime instead of being given before the execution. Second, the partial order among events, not just the set of events, is computed. Therefore, our approach combines the static analysis with the dynamic computation for more precise and comprehensive results. The static analysis produces conservative structural information of the program, using a forward analysis algorithm; and the dynamic computation is based on an efficient vector clock algorithm. It is worth noting that program slicing technique can actually be used to improve our approach by conservatively removing irrelevant program points before hand. We leave this direction to the future work.

Control scope Before we can technically discuss the different types of dependence, we need to define the important and, up to our knowledge, novel concept of *control scope* of a statement. This can be computed entirely statically by examining the structure of the program and using additional information that can be provided by apriory static analysis of the program or via user annotations, such as termination of loops (or recursive functions). While calculating the control scope of statements may look like an interesting problem by itself, its motivation was fully derived from our need to extract very relaxed, yet sound, causal partial orders *at runtime*. We believe that the concept of control scope will play a key role in future endeavors on predictive runtime analysis; however, we cannot see applications of it in purely static analysis settings.

Our goal next is to define the *scope* function mapping a statement L into a set of statements whose execution is decided by the choice made by L; if L is not a choice statement then *scope*(L) is $\emptyset$. For example,

in Figure 1, the choice made at L_{21} decides whether L_{22} and L_{23} are executed or not, but does not affect the execution of L_{24}. So $scope(L_{21}) = \{L_{22}, L_{23}\}$ and $scope(L_{11}) = scope(L_{22}) = \emptyset$. The situation is more complex in the context of possibly non-terminating loops and of control-intensive statements, such as exception. The *scope* function provides precious information regarding the control-flow structure of the program, and can be computed statically form the control-flow graph. We assume the program well structured, meaning that every loop in the control flow graph has only one entry.

Definition 3. *Let $G = \langle V, E, n_s, n_e, \phi \rangle$ be a **control flow (directed) graph**, where V is a set of nodes corresponding to statements in the program and E is a set of edges corresponding to transitions. n_s is the entry node of G and n_e is the exit node of G. Loops in G have only one entry and $\phi \subseteq V$ is a set of (loop) nodes annotated as "terminating".*

Figure 3 shows some examples of control flow graphs. ϕ marks which loops terminate and is used to improve the predictive power of our techniques. It can be obtained either by static analysis or by comments given by programmers. Loops which cannot be statically shown to terminate or which are not annotated so by users are conservatively assumed not to terminate.

Definition 4. *A **complete path** π in G starts with n_s and is either finite and ends with n_e, or infinite and $inf(\pi) - \phi \neq \emptyset$, where $inf(\pi)$ gives those nodes visited infinitely often in π. Let Π denote the **set of complete paths** of G. If $N \subseteq V$ then let Π_N denote the set of complete paths visiting each node in N.*

$\pi \in \Pi$ represents a complete execution path of the program, which either terminates or not. If not, then it must contain at least a non-terminating loop. For Figure 3 (A), $(C_1, L_1, L_3, C_2, L_4, L_6)$ is a complete path and Π contains four complete paths. If $\pi \in \Pi_N$ then π has one or more occurrences of each node in N; π can obviously contain other nodes as well.

Definition 5. *If $n_1, n_2 \in V$ then n_2 **may follow** n_1, written $n_1 \rightsquigarrow n_2$, iff $\Pi_{n_1,n_2} \neq \emptyset$ and for any $\pi \in \Pi_{n_1,n_2}$, there is some n_1 that occurs before any n_2 in π. For a node n_1, we let $mayFollow(n_1)$ be the set $\{n_2 \mid n_1 \rightsquigarrow n_2\}$.*

Note that if $n_1 \rightsquigarrow n_2$ then an occurrence of n_1 does not necessarily imply a subsequent occurrence of n_2; for example, n_1 can be a conditional statement and n_2 resides in a branch of n_1. However, if both n_1 and n_2 occur in a path, once or more times, then at least one of n_1's occurrences must appear before any of n_2's occurrences. Therefore, if $n_1 \rightsquigarrow n_2$ then once n_1 occurs in a path then n_2 may or may not occur, but if it occurs it must follow n_1. In Figure 3 (A), $mayFollow(C_1) = \{L_1, L_2, L_3, C_2, L_4, L_5, L_6\}$.

Proposition 1. *$\rightsquigarrow$ is a strict partial order over the nodes in G.*

Proof. $\rightsquigarrow$ is anti-symmetric because in any path π, only one of n_1 and n_2 can occur first. The transitivity of $\rightsquigarrow$ can be proved as follows. Suppose that $n_1 \rightsquigarrow n_2 \rightsquigarrow n_3$. To show that $n_1 \rightsquigarrow n_3$, we need to first show that $\Pi_{n_1,n_3} \neq \emptyset$ and then that for any $\pi \in \Pi_{n_1,n_3}$ there is some occurrence of n_1 that appears before any occurrence of n_3. Since $n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow n_3$, it follows that there are some complete paths $\alpha_1 n_1 \beta_1 n_2 \gamma_1$ and $\alpha_2 n_2 \beta_2 n_3 \gamma_2$ in Π_{n_1,n_2} and Π_{n_2,n_3}, respectively. Then the path $\alpha_1 n_1 \beta_1 n_2 \beta_2 n_3 \gamma_2$ obviously is in Π_{n_1,n_3}, so $\Pi_{n_1,n_3} \neq \emptyset$. Suppose now, by contradiction, that there is some path $\pi = \alpha n_3 \beta n_1 \gamma$ such that its finite prefix α contains no n_1. Then note that π has no occurrence of n_2, because $n_2 \rightsquigarrow n_3$ implies that such an n_2 would appear in α, which would contradict $n_1 \rightsquigarrow n_2$. Since $n_1 \rightsquigarrow n_2$ there is some path $\pi_1 = \alpha_1 n_1 \beta_1 n_2 \gamma_1$ in Π_{n_1,n_2}. Then we can build the path $\pi' = \alpha n_3 \beta n_1 \beta_1 n_2 \gamma_1$ by concatenating the prefix up to n_1 in π with the suffix following the first n_1 in π_1. Note that $\pi' \in \Pi_{n_2,n_3}$ but n_3 appears before n_2 in it, which contradicts $n_2 \rightsquigarrow n_3$. Therefore, the $\rightsquigarrow$ relation is transitive. □

Definition 6. *If $n_1, n_2 \in V$ then n_2 **must follow** n_1, written $n_1 \curvearrowright n_2$, iff $\Pi_{n_1} \subseteq \Pi_{n_2}$ and $n_1 \rightsquigarrow n_2$. For a node n_1, we let $mustFollow(n_1)$ be the set $\{n_2 \mid n_1 \curvearrowright n_2\}$.*

If $n_1 \curvearrowright n_2$ then n_1 occurs first and n_2 will surely occur later. Obviously, $mustFollow(c) \subseteq mayFollow(c)$ for any $c \in V$. In Figure 3 (A), $mustFollow(C_1) = \{L_3, C_2, L_6\}$. We can prove that $mustFollow(c)$ is a chain w.r.t. to $\rightsquigarrow$ and have the following:

Proposition 2. *For any node $c \in V$, $mustFollow(c) \neq \emptyset$ iff there exists a unique smallest node in $mustFollow(c)$ w.r.t. $\rightsquigarrow$.*

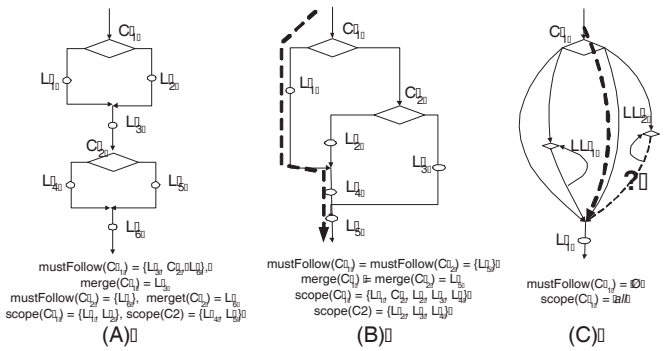

Fig. 3. Examples for computing scope() function

```
      try {
C1 :  :  if (a > 0) then
L1 :  :  : x = x + 1;
      :  else
C2 :  :  : if (b > 0) then
L2 :  :  : : x = x + 2;
      :  : else
L3 :  :  : : y = 1;
C3 :  :  : : throw (exception);
      :  : endif
      :  endif
L4 :  :  z = x;
      } catch (exception);
L5 : y = y + 1;
```

Fig. 4. Program with complex control-flow

Proof. If a smallest node exists, *mustFollow*(c) is surely not empty. To prove that a smallest node exists and is unique when *mustFollow*(c) is not empty, it suffices to show that *mustFollow*(c) is a chain w.r.t. $\rightsquigarrow$. Suppose *mustFollow*(c) is not a chain, which means that there are $n_1, n_2 \in follow(c)$ such that neither $n_1 \rightsquigarrow n_2$ nor $n_2 \rightsquigarrow n_1$. Since $\Pi_{\{n_1,n_2\}} \neq \emptyset$, there are some paths π_1 and π_2 such that n_1 occurs first in π_1 and n_2 occurs first in π_2. That is to say, n_1 and n_2 are within a loop, and the loop has at least two entries: n_1 and n_2. This is impossible since G is was assumed to be well structured, so each cycle has exactly one entry point. □

Definition 7. *If $c \in V$ such that mustFollow$(c) \neq \emptyset$, then let merge(c) be the smallest node in mustFollow(c) w.r.t. $\rightsquigarrow$.*

In Figure 3 (A), *merge*$(C_1) = L_3$. An important question is when *merge*(c) exists, or in other words, when *mustFollow*$(c) \neq \emptyset$. If c is not a choice point then it always has a definite next statement, which will obviously be *merge*(c). Also, if any path in Π_c terminates then n_e occurs in each such path, so *mustFollow*$(c) \neq \emptyset$. Therefore, the only possibility for *merge*(c) not to exist is that c is a choice node and at least one of its branches admits some non-terminating complete path. This problem will be re-discussed shortly. We are now ready to introduce our core concept:

Definition 8. *For any node $c \in V$, let scope(c) be the set of nodes mayFollow(c) − mayFollow(merge(c)) − merge(c). In other words, scope(c) is the set of nodes n with $c \rightsquigarrow n$ and, if merge(c) exists, with $n \rightsquigarrow$ merge(c).*

Note that *scope*$(c) = \emptyset$ for non-choice nodes. For choice nodes, *scope*(c) contains the statements in both branches, but not those following and including their merge points. Figure 3 (A) shows a simple example. Also, note that *merge*(c) does not exist (or in other words *mustFollow*$(c) = \emptyset$) iff *scope*(c) = *mayFollow*(c).

Proposition 3. *If $c \in V$ then mustFollow$(c) = \emptyset$ iff scope(c) contains some non-terminating loop node (one not in ϕ).*

Proof. If *mustFollow*$(c) = \emptyset$, or in other words if *scope*(c) = *mayFollow*(c), then suppose that *scope*(c) contains only terminating loop nodes in ϕ; that means that any $\pi \in \Pi_c$ contains n_e, so $n_e \in$ *mustFollow*(c), which is impossible. Conversely, suppose that *scope*(c) contains some non-terminating loop entry node n and that *mustFollow*(c) is not empty. Because of the well-structuredness of G, the loop of n cannot contain any node in *mustFollow*(c). Therefore, we can get a complete path π which ends with the endless loop of n and which does not contain any node in *mustFollow*(c). This breaks the definition of *mustFollow*(c). So *mustFollow*(c) is empty.

Let us next focus on calculating the scope function. As mentioned above, for a choice node c, *scope*(c) contains the statements in both its branches, but not those following and including its merge point. The situation becomes more complicated in the presence of control sensitive statements, such as *exceptions*. Figure 3 (B) shows an example, corresponding to the code in Figure 4. The branches of C_1 contain statements C_2 and L_1, while the branches of C_2 contain L_2 and L_3. However, L_4, despite the fact that it is outside the syntactic boundaries of the conditional C_2, it actually *depends* upon the decision taken by C_2, because the other branch could have potentially changed the control flow by throwing an exception, so L_4 exists because C_2 did not choose the other branch. Since C_2 depends on C_1, L_4 also depends on C_1. One may consider solving such relation at runtime to keep the static information minimum, using the transitivity of

the dependence: the execution of L_4 depends on the choice of C_2 and C_2 depends on C_1. However, structural information may be missing at runtime. Think of an execution choosing the branch containing L_1, as the path depicted by the heavy dotted line in Figure 3 (B) shows, the dependence from L_4 to C_2 is lost since C_2 is not executed. Then the runtime analysis may show that there is no dependence from L_4 to C_1. So the algorithm needs to assure that $scope(C_2) \subseteq scope(C_1)$, if $C_2 \in scope(C_1)$.

Because of non-termination, loops are split into ones which terminate (those in ϕ) and ones which do not. If a loop terminates then the code following it will be executed anyway, so the statements following the loop are *not* in the scope of the loop condition. Therefore, if a loop statement L is in ϕ then *scope(L)* only contains the statements in the loop body; otherwise, *scope(L)* contains both the statements in the loop body and those following the loop. Recall that the statements in $scope(L)$ are those whose execution is conditioned by the particular choice made by L. In the case of non-terminating loops, due to our dual static/dynamic approach we actually do not need to perform a reachability analysis to find all the statements following the loop conditional; that is because any statement reached at runtime is certainly reachable. Therefore, we introduce a special (symbolic) set "all", representing all the reachable statements, and set $scope(c)$ to all.

There is a complex situation regarding loops that can be, fortunately, surprisingly easily dealt with. Suppose that some loops exist in some paths from a conditional statement to its corresponding merge point, as shown in Figure 3 (C). There are two loops on two paths from C_1 to L_1, namely LL_1 and LL_2. The dotted line with a question mark means that LL_2 may not terminate, and the heavy dotted line shows an actual execution. The tricky aspect is that if one of the paths controlled by C_1 is non-terminating, whether the statements following the conditional will be executed or not depends upon the decision made at C_1, even though the actual execution terminates. That is to say, if all loops on the paths controlled by C_1 terminate, then $scope(C_1)$ contains all the statements in all its branches; otherwise, $scope(C_1) = \mathsf{all}$. Assuming that $\mathsf{all} \subseteq \mathsf{all}$, we just need to ensure $scope(L) \subseteq scope(C)$ for all $L \in scope(C)$. *continue* and *break* statements are handled like the exceptions.

Figure 5 puts all these together into an efficient algorithm to compute $scope(L)$. Let us apply this algorithm on the program in Figure 4. First, $computeScope(C_1)$ is called, which in calls $computeScope(L_1)$ and $computeScope(C_2)$. $computeScope(L_1) = \emptyset$. $computeScope(C_2)$ calls $computeScope(C_3)$. The scope of C_3 contains all statements between C_3 and the catch statement, that is $\{L_4\}$. So $computeScope(C_2) = \{L_2, L_3, L_4\}$. Finally we get $computeScope(C_1) = \{L_1, C_1, L_2, L_3, L_4\}$.

Interestingly, thanks to our dual dynamic/static approach, we do not need to extend the scope procedure statically to deal with function calls. The reason for which we need the scope function later on is to test whether the statement L' producing an event e' is in the scope of the statement L that produced a previous event e. This suggests that function call statements occurring in $scope(c)$ can be processed *at runtime*, on a by-need basis: if L is a function call in $scope(c)$ that is reached during the actual execution of the program, then add all the statements reachable from L symbolically to $scope(c)$ (this is similar to "all"). By taking care of function calls dynamically, we can reduce the complexity of static analysis. Returning from functions is handled same way as exceptions; throwing and catching exceptions outside functions is treated as returning and testing special values. Like loops, function calls may also not terminate. Therefore, like for loops, we reside on static analysis for termination or user annotations stating so for each function and collect this information in ϕ.

```
Global variable: L, a statement list sorted by line numbers
procedure computeScope(L)
1.  scope(L) = ∅
2.  L = L − L
3.  switch (type of L)
4.    case (if statement)
5.      for all L' in branches of L do
6.        computeScope(L')
7.        scope(L) = scope(L) ∪ {L'} ∪ scope(L')
8.      endfor ; break
9.    case (while statement)
10.     if (L terminates)
11.     then
12.       for all L' in the body of L do
13.         computeScope(L')
14.         scope(L) = scope(L) ∪ L' ∪ scope(L')
15.       endfor
16.     else
17.       scope(L) = all
18.     endif ; break
19.   case (jump statement)
20.     for all L' between L and its target do
21.       computeScope(L')
22.       scope(L) = scope(L) ∪ L' ∪ scope(L')
23.     endfor ; break
24. endswitch
25. endif
endprocedure
```

Fig. 5. Algorithm to compute the scope function

Control-flow Dependence. If a change of $state(e)$ may affect the occurrence of e', then we say that e' has a *control-flow dependence* on e, and write $e \sqsubset_{ctrl} e'$.

Definition 9. *Let $<$ denote the union of the total orders on events of each thread, i.e., $e < e'$ iff thread(e) = thread(e') and $e <_\tau e'$.*

This relation is extended by convention to abstract relevant events (when these are defined): if $e < e'$ then we also write $\alpha_\varphi(e) < e'$ and $e < \alpha_\varphi(e')$ and $\alpha_\varphi(e) < \alpha_\varphi(e')$. Then, with the help of the *scope* function discussed above, we can define the control-flow dependence on events as follows:

Definition 10. *We write $e \sqsubset_{ctrl} e'$ iff $e < e'$ and stmt$(e') \in$ scope(stmt(e)), and e' is smallest with this property, i.e., there is no e'' such that $e < e'' < e'$ and stmt$(e') \in$ scope(stmt(e'')).*

In other words, if e and e' are events occurring within the same thread in an execution trace τ of some multi-threaded system, we say that e' has a *control-flow dependence* on e, written $e \sqsubset_{ctrl} e'$, iff e is the *latest* event occurring before e' with the statement that generated e' in the control scope of the statement that generated e. Hence, the control dependence relation has a combined dynamic/static flavor in our approach. For the example trace in Figure 2, it is easy to see that $e_2 \sqsubset_{ctrl} e_5, e_6, e_7, e_8$. This control-flow dependence extends by convention to abstract relevant events (when defined) as expected: if $e \sqsubset_{ctrl} e'$ then $e \sqsubset_{ctrl} \alpha_\varphi(e')$, $\alpha_\varphi(e) \sqsubset_{ctrl} e'$, and $\alpha_v arphi(e) \sqsubset_{ctrl} \alpha_\varphi(e')$.

We want to be able to show that the existence of an event e is determined by the existence of all the events e' with $e' \sqsubset_{ctrl} e$. To distinguish among different occurrences of events with the same attribute values, let us add a new attribute to every event, *counter*, collecting the number of previous events with the same attribute-value pairs (other than the *counter*). Event e is said to *occur* in a partial trace β iff there is an event e_{abs} in $\alpha_\varphi(\beta)$, such that for any attribute *key*, either *key*(e) = *key*(e_{abs}) or both are undefined. Event e is said to *occur regardless of attribute key* in β iff there is some e_{abs} in $\alpha_\varphi(\beta)$, such that for any attribute *key'* other than *key*, either *key'*(e) = *key'*(e_{abs}) or both are undefined. Suppose an incomplete execution of the program that generated partial trace β and a relevant event e that has not occurred yet but has *counter(e)* -1 occurrences regardless of state in β. Also, suppose that for any event e' with $e' \sqsubset_{ctrl} e$, e' has already occurred in β. Then we claim that e will occur regardless of its *state* and *counter* when the execution continues, independently of thread scheduling choices. The detailed formalization of these intuitions seems technically intricate and probably not worth the effort.

Data-flow Dependence. If a change of *state*(e) may affect the *state*(e') then we say e' has a *data-flow dependence* on e and write $e \sqsubset_{data} e'$.

Definition 11. *For two events e and e', $e \sqsubset_{data} e'$ iff $e <_\tau e'$ and one of the following three situations happens: (1) $e < e'$, type(e) = read and stmt(e') uses target(e) to compute state(e'); (2) type(e) = write, type(e') = read, target(e) = target(e'), and there is no other e'' with $e <_\tau e'' <_\tau e'$, type(e'') = write, and target(e'') = target(e'); (3) $e < e'$, type(e') = read, stmt$(e') \notin$ scope(stmt(e)), and there exists a statement S in scope(stmt(e)) s.t. S can change the value of target(e').*

In the first case, e and e' are generated by the same thread, and e is a read of some variable x whose value is used by the statement generating event e' to compute *state*(e'); this case makes sense when e' is a write event or a function call whose actual arguments refer to x. For example, in Figure 2, $e_5 \sqsubset_{data} e_6$. The second case states that a read of a shared variable depends on the latest write of it. For example, in Figure 2, $e_1 \sqsubset_{data} e_2$. Note that the two events can be generated by different threads. The third case is the most subtle one and states that a read of a variable may depend on preceding control-flow choices. Consider the example in Figure 4: the value of y at L_5 depends upon the value of a at C_1, even though L_3 is not executed! That is because the value of y read at L_5 could have been different if C_1 made a different choice. A simple structural analysis of the program gives the computational dependence needed for the first case. Also, it is straightforward to retrieve from τ the write-read dependence needed for the second case. However, it is hard to decide the dependence between a read and the preceding control flow, because it relies on the fact that unexplored branches may have changed the value of the location to read. To achieve this, one needs to perform a *static alias analysis*, an undecidable process in general. To avoid false alarms, we recommend a conservative alias analysis giving "may have" alias partitions of variable accesses.

To seamlessly incorporate function calls, we create a special variable *_result* to represent the return value of functions. When a function returns, a write on *_result* is generated. When the return value is used, a read on *_result* is generated. This way, dependences caused by expanding function calls are handled at no additional costs.

Note that there are no write-write, read-read, read-write data dependences. Case (2) above only considers the write-read data dependence. However, it enforces the read to depend upon only the latest write of the same variable; this way, a write and the following reads of the same shared variable form an *atomic* block of events. This captures entirely the work presented in [16], but in the much more general setting of this paper.

Similarly to the control-flow dependence, the data-flow dependence also extends by convention to abstract relevant events (when defined) as expected: if $e \sqsubset_{data} e'$ then $e \sqsubset_{data} \alpha_\varphi(e')$, $\alpha_\varphi(e) \sqsubset_{data} e'$, and $\alpha_v archi(e) \sqsubset_{data} \alpha_\varphi(e')$. One can now show that an event e is uniquely determined by the all the events e' with $e' \sqsubset_{data} e$. Suppose an incomplete execution of the program that generated partial trace β and a relevant event e that has not occurred yet but which will occur regardless of its *state* attribute, which also has the property for any event e' with $e' \sqsubset_{data} e$, e' has already occurred in β. Then e (including the value of its *state*) will also occur when this execution continues, independently of thread scheduling. Note that if the abstract event e does not contain a state attribute, then the data-flow dependence is not taken into account.

2.3 Causal Partial Orders and Sound Permutations

We are now ready to define our novel notion of control-flow and data-flow dependence on events:

Definition 12. *Event e **depends upon** e' iff $e \sqsubset e'$, where $\sqsubset$ is the relation $(\sqsubset_{data} \cup \sqsubset_{ctrl})^+$.*

Based on the discussions at the end of the previous two subsections, one can show that the preservation of dependence guarantees the occurrence of relevant events and also preserves their state. Our major goal is to *generate and analyze permutations of relevant events* that correspond to possible executions of the system.

Definition 13. *A permutation of ξ_φ is **sound** iff there is some execution whose trace can be abstracted to this permutation.*

The appealing aspect of predictive runtime analysis is that *one does not need to re-execute the program* to generate sound traces; instead, we define an appropriate notion of *causal partial order* and then prove that any permutation consistent with it is sound. Intuitively, a sound permutation preserves relevant events as well as events upon which relevant events depend.

Definition 14. *Let $\overline{\xi_\varphi} \subseteq \xi \cup \xi_\varphi$ be the set extending ξ_φ with events $e \in \xi$ s.t. $e \sqsubset e'$ for some $e' \in \xi_\varphi$. We then let $\prec \subseteq \overline{\xi_\varphi} \times \overline{\xi_\varphi}$ be the **causal partial order** relation defined as $(< \cup \sqsubset)^+$.*

Therefore, the causal partial order is nothing but the dependence relation extended with the total order on the events generated by each thread. The causal partial order was defined on more events than those in ξ_φ, but in order to generate sound permutations of relevant events we only need its projection onto the relevant events:

Theorem 1. *A permutation of ξ_φ is a sound abstract trace whenever it is consistent with the above causal partial order.*

Proof. Let $e_1 e_2 ...$ be a permutation of the events in ξ_φ that is consistent with $\prec$, or in other words a *linearization of* $\prec$, and let $\Sigma_i = \{e_1, ..., e_i\}$ denote the set of the first i events of this abstract trace. Then one can easily show by induction on i that if $e \prec e_i$ for some event e, then $e \in \Sigma_i$. Such sets Σ_i are also called *consistent cuts* and will be further discussed in Section 3.2. Then we can construct an abstract execution of the program for this permutation by induction (same steps are followed to generate a counter-example when the property is violated):

1. For e_1, we simply start the thread *thread*(e_1) and pause it after e_1 is generated;
2. For e_i, by the induction hypothesis we have constructed an execution of the program which produces $e_1 ... e_{i-1}$. Since all the events upon which e_i depends are already preserved in the execution, we can safely start the thread *thread*(e_i) to produce e_i and pause it.

We can therefore simulate an execution of the system that generates the original permutation of relevant events as an abstract trace.

3 Generating Sound Permutations

The section describes the generation of sound permutations based on the dependence-based causal partial order. First, a vector clock (VC) based algorithm that computes the causal partial order among relevant events is introduced. Then we show that the causal order can be further relaxed by considering the lock-related atomicity. At the end, a generic algorithm is illustrated, which generates sound permutations of relevant events level by level.

3.1 Computing Causal Partial Order

We extend the VC-based algorithm in [15] to support dependence among events in our approach.

Definition 15. *A vector clock (VC) is a function from threads to integers, $VC : T \rightarrow Int$, where T is the set of threads. $VC \leq VC'$ iff $\forall t \in T, VC(t) \leq VC'(t)$. And we have the max function for VC: $max(VC_1, ..., VC_n)(t) = max(VC_1(t), ..., VC_n(t))$.*

Every thread t has a VC_t, which keeps the order within the thread as well as the information about other threads that it knows from their communication (read/write on shared variables). Every variable x has a VC_x that shows how the value of the variable is computed. And every shared variable var has a VC^r_x that accumulates the information about variable accesses. When a relevant event e is encountered, it will be associated with a VC_e, which encodes the causal partial order. We next show how to update these VCs when an event e is encountered during the analysis. Note that the third case can overlap the first two cases.

1. $type(e) = write, target(e) = x, thread(e) = t$ (the variable x is written in thread t). In this case, the VC_x is updated using VC^r_x, VC_t, and VCs of those events upon which e depends: $VC_x = max(VC^r_x, VC_t, VC_{e1}, ..., VC_{en})$ where $e_1, ..., e_n \sqsubset e$. Then $VC_e = VC^r_x = VC_x$.
2. $type(e) = read, target(e) = x, thread(e) = t$ (the variable x is read in t), and x is a shared variable. The information of the thread is accumulated into VC^r_x: $VC^r_x = max(VC^r_x, VC_t)$, and $VC_e = VC^r_x$.
3. e is a relevant event, where $thread(e) = t$. For this case, VC_t needs to be increased in order to keep the total order within the thread, and the event will be issued to the observer with an up-to-date VC. However, as discussed below, some previous events that were regarded as irrelevant events become useful due to the occurrence of e. This requires backtracking to refine VCs of some previous relevant events. Suppose $e_1, ..., e_n \sqsubset e$; for any i, $1 \leq i \leq n$, if $VC_{ei} \not\leq VC_t$, then for any processed relevant events e', where $VC_{e'} \not\leq VC_{ei}$, let $VC_{e'} = max(VC_{e'}, VC_{ei})$. And let $VC_t(t) = VC_t(t) + 1$, then $VC_t = max(VC_t, VC_{e1}, ..., VC_{en})$, $VC_e = VC_t$.

Figure 6 (A) and (B) illustrate two cases that require backtracking in the above algorithm, in which $e, e', e'' \in \xi_\varphi$ and $e_1, e_2 \notin \xi_\varphi$. Basically, this is caused by some "delayed" dependence among events. For the case in (A), when e' is processed, e_1 seems irrelevant and is not taken into account by the algorithm. But when e'' is encountered, e_1 becomes a useful event and the algorithm has to re-compute $VC_{e'}$ in order to achieve a correct result. (B) is similar but a little more complex: e_1 and e_2 are considered irrelevant until e'' is hit. To recognize such cases, we can notice that, if e_1 is not taken into account, the thread's vector clock is not updated using VC_{e1}. Therefore, we have $VC_{e1} \not\leq VC_t$. And for the same reason, if e' has been processed and $e_1 < e'$, $VC_{e1} \not\leq VC_{e'}$. This way, we can easily go back and refine the VCs of previous events.

Because of the backtracking, in the worst case, the time cost of the above algorithm is $O(|\xi|^2)$. It is worth noting that the backtracking is only needed for calculating the vector clock online, which means that only the information up to some point is available and the result has to be refined gradually when new information is learned. For offline analysis, i.e. the analysis is carried out after the execution finishes, on the other hand, one can first scan the obtained trace backwards to figure out all useful events and then compute VCs from the beginning, reducing the time cost to $O(\xi)$. However, the online version of the algorithm is adopted in our prototype, although it presently works in the offline mode, because the experiments show that in practice the backtracking rarely happens (Section 5). And we can prove correctness of this algorithm.

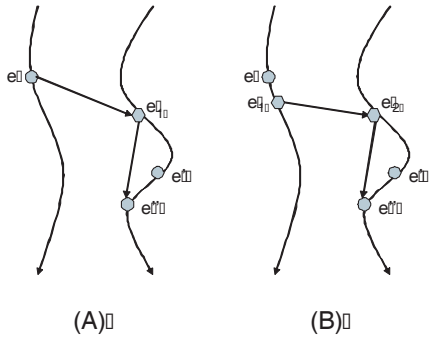

Fig. 6. Backtracking cases for VC Generation

Theorem 2. $e \prec e' \Rightarrow VC_e \leq VC_{e'}$

Proof. The algorithm to compute VCs is derived from the one discussed in [16], for which the correctness has been proved. The extension here is that the dependence is taken into account when computing the VCs of variables and relevant events. Using the max function, we have that $e \sqsubset e' \Rightarrow VC_e \leq VC_{e'}$. Based on this consideration, it is straightforward to prove this theorem using the previous proof.

Note that $\leq$ among VCs is stronger than $\prec$ among events. This is because when VCs are computed, the read-after-write order is also taken into account (the second case), which the $\prec$ order does not have to keep, as discussed in Section 2.2. Combining with Theorem 1, we can get the following proposition immediately:

Proposition 4. *The permutation consistent with $\leq$ among events' VCs is sound.*

3.2 Lock-Atomicity of Events

One may further loosen the causal partial order if more semantic information is obtained for the program. We next discuss how to incorporate the lock mechanism into our approach to allow more sound permutations. Locks play a significant role in multi-threaded programs. In most causal order based approaches, locks are treated as shared variables, while acquiring and releasing locks are viewed as accesses to locks. This way, blocks protected by the same lock are naturally ordered and kept exclusive to one another. However, the order caused by the lock is stronger than the actual lock semantics, which only imposes the mutual exclusion among blocks. To better support lock semantics, our approach introduces *lock related atomicity*. Using this concept, two sets of events that are atomic w.r.t. the same lock cannot be interleaved, but can be permuted if there are no causal orders between them.

First, two new types of events are introduced for lock operations, i.e. *acquire* and *release*. The target of such kinds of events is the lock to be accessed. If there are embedded lock operations on the same lock(i.e. a thread can acquire the same lock multiple times), only the outmost acquire-release pair generates the corresponding events. The control-flow dependence is extended correspondingly:

Definition 16. *$e, e' \in \xi, type(e) = acquire, e \sqsubset_{ctrl} e'$ iff $e < e' < e''$, where e'' is the corresponding release of e.*

That is to say, an event e protected by an acquiring of l has the control-flow dependence on the acquiring event. Two events protected by the same lock are *atomic w.r.t. the lock*:

Definition 17. *Two events e and e' are* l-atomic*, denoted by $e \Updownarrow_l e'$, iff $\exists e'' \in \xi, type(e'') = acquire, target(e'') = l$, $e'' \sqsubset_{ctrl} e$ and $e'' \sqsubset_{ctrl} e'$. $\Updownarrow_l$ is an equivalence relation on ξ_φ. Let $[e]_l$ denote the corresponding equivalence class of an event $e \in \xi_\varphi$.*

Every sound permutation can be viewed as an abstract run of the program. A run is called consistent if it not only follows the causal order, but also preserves the lock-atomicity of events. Let us first define the concept of *consistent cuts*:

Definition 18. *A cut Σ is a set of events. Σ is consistent iff for all $e, e' \in \Sigma$,*
(a) if $e \in \Sigma$ and $e' \prec e$, then $e' \in \Sigma$ and
(b) if $e \notin [e']_l$ for some lock l, then either $[e]_l \subseteq \Sigma$ or $[e']_l \subseteq \Sigma$.

The first property shows that for any event in Σ, all the events upon which it depends should also be in Σ. The second property states that there is at most one incomplete l-atomic set in Σ. Otherwise, the l-atomicity is broken. Essentially, Σ contains the events in the prefix of a consistent run. When an event e can be added to Σ without breaking the consistency, e is called *enabled* for Σ.

Definition 19. *An event e is enabled for a consistent cut Σ iff*
(a) for any event $e' \in \Sigma$, if $e' \prec e$, then $e' \in \Sigma$, and
(b) for any event $e' \in \Sigma$ and any lock l, either $e \in [e']_l$ or $[e']_l \subseteq \Sigma$.

This definition is equivalent to the following one:

Definition 20. *e is enabled for a consistent cut Σ iff $\Sigma \cup \{e\}$ is also consistent.*

Now we can define a consistent run:

Definition 21. *A* consistent multi-threaded run *$e_1 e_2 ... e_{|\xi_\varphi|}$ generates a sequence of consistent cuts $\Sigma_0 \Sigma_1 ... \Sigma_{|\xi_\varphi|}$, such that for all $1 \leq r \leq |\xi_\varphi|$, Σ_{r-1} is a consistent cut, e_r is enabled for Σ_{r-1}, and $\Sigma_r = \Sigma_{r-1} \cup \{e_r\}$.*

Theorem 3. *A consistent run of ξ_φ is sound.*

Proof. Similar to the proof for Theorem 1, the definition of the consistent run actually gives the way to construct an execution of the program which can be represented by the permutation.

3.3 Capturing Lock-Atomicity

To capture the lock-atomicity among events, we associate a counter $counter_l$ with every lock l. Let LS_t denote the set of locks held by the thread t. A new attribute, LS, is also added into the event, whose value is a mapping on locks to corresponding counters. When an event e is processed, the lock information is updated as follows:

1. if $type(e) = acquire, thread(e) = t, target(e) = l$, then $counter_l = counter_l + 1$, $LS_t = LS_t \cup \{l\}$.
2. if $type(e) = release, thread(e) = t, target(e) = l$, then $LS_t = LS_t - \{l\}$.
3. if $\alpha(e)$ defined, then let $LS(e)(l) = count_l$ for any l in $LS_{thread(e)}$, and $LS(e)(l) = -1$ for any other l.

We can have the following theorem for the correctness of this algorithm. The proof is straightforward, so we ignore it here.

Theorem 4. *$e \Updownarrow_l e'$ iff $LS(e)(l) = LS(e')(l) \neq -1$*

3.4 Level by Level Generation of Consistent Runs

Figure 7 gives the algorithm to generate and verify consistent runs based on the causal partial order and the lock-atomicity level by level. In this algorithm, ξ_φ is the set of relevant events, while $CurrentLevel$ and $NextLevel$ are sets of cuts. We do not store all the events for the cut Σ in the algorithm, instead, Σ is represented using the following information: the VCs of threads and shared variables, lock sets held by threads, and the current state of the property monitor for this run. The property monitor is a program which verifies the run against the desired property. In our approach, the monitor is automatically generated from the specification of the property (see Section 4).

```
procedure main()
1.  while (ξ_φ ≠ ∅)
2.  ⋮ verifyNextLevel()
3.  endwhile
endprocedure
procedure createCut(Σ, m, ξ_φ)
1.  if not monitor(Σ, m)
2.  then
3.  ⋮ reportViolation(Σ, m)
4.  ⋮ return
5.  endif
6.  Σ' ← new copy of Σ
7.  i ← thread(m)
8.  VC(Σ')[i] ← VC(Σ)[i] + 1
9.  if type(m) = acquire and target(m) = l
10. then
11. ⋮ LS_l(Σ) ← LS_l(m)
12. else
13. ⋮ if type(m) = release and target(m) = l
14. ⋮ then
15. ⋮ ⋮ LS_l(Σ) ← −1
16. ⋮ endif
17. endif
endprocedure

procedure verifyNextLevel()
1.  for all m ∈ ξ_φ and Σ ∈ CurrentLevel do
2.  ⋮ if enabled(Σ, m)
3.  ⋮ then
4.  ⋮ ⋮ NextLevel ← NextLevel ∪ createCut(Σ, m, ξ_φ)
5.  ⋮ endif
6.  ⋮ if w > 0 and size(NextLevel) > width
7.  ⋮ then break endif
8.  endfor
9.  ξ_φ ← removeUselessMessages(CurrentLevel, ξ_φ)
10. CurrentLevel ← NextLevel
11. NextLevel ← ∅
endprocedure
procedure enabled(Σ, m)
1.  i ← thread(m)
2.  if not (∀j ≠ i : VC(Σ)[j] ≥ V[j])
    ⋮ ⋮ ⋮ and VC(Σ)[i] + 1 = V[i]
3.  then return false endif
4.  if (∃lock l, LS_l(m) > −1, LS_l(Σ) > −1,
    ⋮ ⋮ ⋮ and LS_l(Σ) ≠ LS_l(m))
5.  then return false endif
6.  return true
endprocedure
```

Fig. 7. Consistent runs generation algorithm

The algorithm first checks every event in ξ_φ and every cut in the current level to generate cuts of the next level by appending enabled events to the current cuts. The user can set the width of every level in order to control the state space to search. After the next level is generated, redundant events, which are already

processed in all runs, will be removed from ξ_φ. The *enabled* procedure essentially implements the definition of the consistent run. If an event e is enabled for a cut Σ, it will be sent to the monitor along with Σ to be verified against the desired property. Violation is reported once detected. Otherwise, e is applied to Σ to create a new cut.

4 Implementation

We have implemented a prototype tool to apply this approach on multi-threaded Java programs. The architecture of the prototype is shown in Figure 8. The tool is composed of three parts, including a static analyzer, a trace analyzer and a monitor synthesizer. The monitor synthesizer is mainly based on the work of the Java-MOP tool [2]. Briefly, it reads in the property specification, generates monitoring code that will check the event trace against the property, and also provides definitions of relevant events to the trace analyzer. The monitor synthesizer is extensible w.r.t. the specification formalism. For the time being, it supports the specifications written in linear temporal logic (LTL) and extended regular expressions (ERE). Interested readers can refer to [1] for discussion about monitor synthesis. In addition to checking temporal properties, a specialized monitor for race condition detection is also implemented.

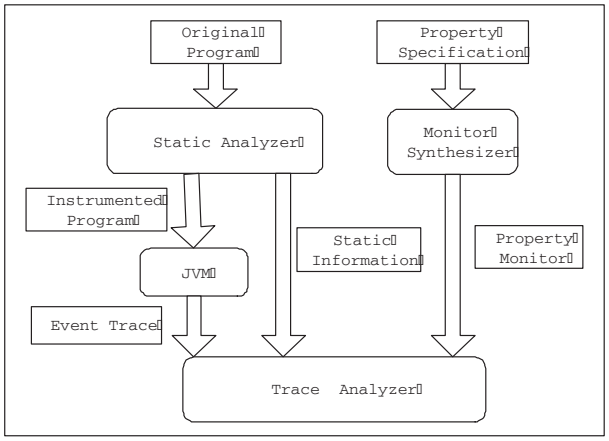

Fig. 8. Analysis components

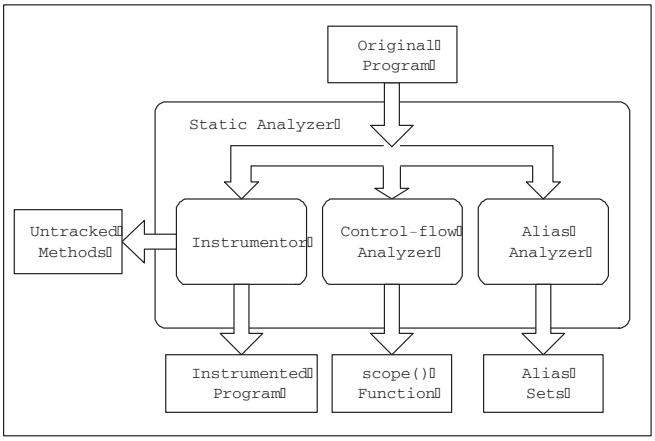

Fig. 9. Architecture of static analyzer

4.1 Static Analyzer

The static analyzer takes the original program as input and produces the instrumented program as well as auxiliary static information about the program. As shown in Figure 9, the static analyzer consists of three components, namely, a program instrumentor, a control flow analyzer and an alias analyzer. All the outputs are stored in ASCII text files that are used later by the trace analyzer.

The program instrumentor is the core component of the static analyzer. It instruments the program with event generating instructions. The instrumented program will produce detailed events for reads/writes on memory locations and begins/ends of function calls. These events are put into a global synchronized buffer and then written to the log file.

To achieve the maximum precision, the trace analysis requires a complete execution trace, which means that all involved methods should be instrumented. In practice, this can cause a huge runtime overhead, and sometimes it is even impossible to achieve, e.g. native methods in Java. Therefore, our tool allows the user to specify the classes to instrument and then produces a list of methods that are not instrumented but invoked in the instrumented program. Actually, the user can control granuality and performance of the analysis by choosing different sets of classes to instrument. The fewer classes are instrumented, the better performance is obtained but some precision can be lost, or vice versus.

To obtain better precision of the analysis, the user needs to annotate those un-tracked methods with purity information. Pure methods do not change the receiver object and will be viewed as reads on the

object, while non-pure methods are regarded as writes on the receiver. Also, those arguments that can be changed in the method need to be annotated as *out* arguments. Such method annotation can be reused. In this sense, the tool could be combined with the work on contract-based approaches, e.g. JML [11], to achieve better results. By default, all un-tracked methods are assumed to be not pure and all arguments of reference types are assumed to be vulnerable. This way, the tool is conservative but allows more false positives.

Another kind of annotation needed is the termination of loops and recursion, as discussed in Section 2.2. In order to reduce the burden on the programmer, our prototype uses a heuristic assumption for loops: when the condition of the loop involves no shared variables, the loop is assumed to terminate. This assumption brings unsoundness into the tool, but turned out to be so effective in evaluation that we do not annotate any loops in experiments. Our prototype has not implemented analysis on recursive calls, but we believe that a similar assumption can be applied.

The control flow analyzer implements the algorithm in Figure 5, while the alias analyzer implements a simple alias analysis in our current prototype, which will be replaced with a more powerful implementation based on some up-to-date research work.

4.2 Trace Analyzer

The implementation of the trace analyzer is shown in Figure 10. Its input includes the execution trace generated by the instrumented program and the static information produced by the static analyzer, along with the monitor to check the desired property. Currently, this prototype works in the offline mode, which means that the trace analyzer is not invoked at runtime. Instead, it is used after the execution to analyze the generated trace log. The analysis process is divided into three phases in order to improve the effectiveness and efficiency of the tool. But the analysis can also be implemented as an online process which may lose some effectiveness.

In the first phase, the pre-processor goes through the input execution trace, collecting information about the usage of objects to mark out shared variables and also record lifecyles of objects. Based on such information, the *VC* generator can precisely compute VCs, and minimize the usage of memory by discarding information about objects when they are dead.

Based on the information obtained from the static analyzer and the pre-processor, the *VC* generator extracts relevant events from the original trace, and computes corresponding VCs and lock-atomicity using the algorithms in Section 3.1 and Section 3.2. Note that since the trace contains detailed runtime information, the analysis is fine-grained, e.g. every element in an array is computed individually. However, in many cases, such fine grained analysis is not necessary and increases performance impact, as shown in Section 5. The optimization based on this observation is currently being implemented.

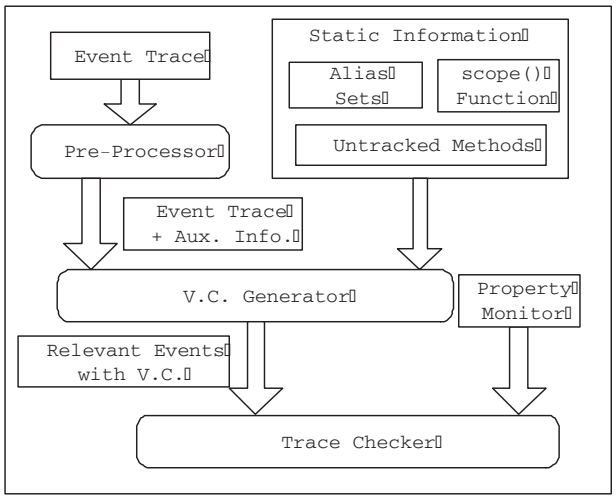

Fig. 10. Architecture of trace analyzer

The relevant event set, along with the relationship represented by vector clocks and lock-atomic sets, is then fed into the trace checker, to verify against the desired property. The trace checker generates consistent runs level by level, using the algorithm in Figure 7, and invokes the property monitor to verify these runs. It is worth noting that for some properties, it is unnecessary to really generate those runs. As an example, we implement a specific checker to detect race conditions. In our algorithm, a race on the shared variable x happens iff $\exists e_1, e_2 \in \xi, target(e_1) = target(e_2) = x$, at least one of them is a write, $VC(e_1) \neq VC(e_2)$, and $\nexists l$, where $LS(e_1)(l) \neq -1$, $LS(e_2)(l) \neq -1$, and $LS(e_1)(l) \neq LS(e_2)(l)$.

Basically, race conditions on different variables are viewed as different properties. In other words, to detect races on another variable, the tool needs to do the VC generation and property verification all over

again because different relevant events are involved. To avoid redundant computation, one optimization is to implement a lock-set algorithm in the pre-processor to remove those well-protected shared variables. Another optimization is to check races on a shared array as a whole instead of on individual elements of the array. Moreover, noticing that the number of locks in a system is comparatively small, we categorize the shared variable accesses based on corresponding locks, and only compare those accesses that do not fall into the same lock category. This way, the performance of the race checker is improved.

5 Experiments

There are two major concerns to investigate with the evaluation of the prototype. The first is the runtime overhead caused by the instrumentation of the original program. The other concern is the effectiveness and efficiency of the analysis. It is easy to show that, because of the backtracking, the time cost of the *VC* generation is $O(n^2)$, where n is the length of ξ, in the worst case. And the time cost of the trace generation is $O(m!)$, where m is the length of ξ_φ, in the worst case. Although both conclusions seems unappealing, our experiments show that our approach can detect concurrency safety bugs with reasonable slowdown and good analysis performance. All these experiments were done on a 2.8GHz Pentium4 machine with a 1 GB memory.

5.1 Benchmarks

Figure 11 shows the benchmarks that we use, along with their slowdown ratios after instrumentation. The *S.V.* column gives the number of shared variables found in the execution. The banking and Http-Server are two simple examples from [20], showing some concurrent bug patterns discussed in [5]. We will give a more detailed discussion of the Http-server example in Section 5.3.

Program	lines	Slowdown	S. V.	Threads
Banking	150	x3	10	11
Http-Server	170	x3	2	7
Daisy	1.3K	x10	312	3
Daisy-2	1.5K	x20	572	3
Raytracer	1.8k	x1000	2K	4
Tomcat	60K	?	431	10

Fig. 11. Benchmarks used in evaluation

Daisy [4] is a small file system that was developed as a concrete system for application software verification tools. It is highly concurrent with fine-grained locking. Specifically, it uses a RandomAccessFile to simulate the hard disk, and spin-wait locks to protect every block and every directory. Since the RandomAccessFile is a native class in Java and cannot be instrumented, the warning produced by our tool is imprecise: it only points out that there are races on the *disk* variable, which is an object of the RandomAccessFile, but does not give more specific reasons.

To better test the tool, we implemented a revised version of Daisy, Daisy-2, which replaces the RandomAccessFile by the PseudoFile class that is based on a byte array. For this version, the tool successfully reports a fine-grained race condition, which is discussed in Section 5.2. Both versions involve a large number of shared variables because every block of the disk holds a shared variable as the mutex lock (see Figure 13), imposing a heavy load on the analysis tool. Daisy-2 has many more shared variables because every element in the byte array is regarded as a shared location.

Raytracer is a program used in the Java Grande benchmark [8], which implements a multi-threaded ray tracing algorithm. Tomcat [19] is a popular open source Java application server. The version used in the experiment is 4.1.27, which is the last version of Tomcat 4. We only instrumented the core classes of Tomcat and some related Apache classes to reduce performance impact.

From the figure, we can see that the runtime overhead caused by the event generation is acceptable for most applications. For Tomcat, there is not an accurate estimation of the slowdown ratio. But the response of the instrumented web server under a light workload does not show any significant delay. The performance of the raytracer is greatly reduced. This is caused by the intensive computation over a large array in the algorithm and also the exhaustive instrumentation of the program. To improve the performance, one may choose to not instrument the class manipulating the array. This way, multiple operations on the array will be collapsed into one operation on the manipulating object, resulting in fewer events and shared variables.

5.2 Detecting Dataraces

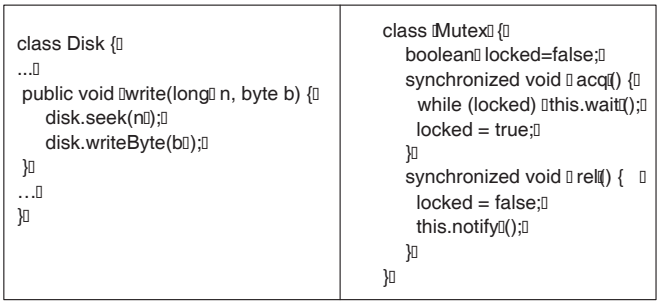

Fig. 13. Implementation of Disk and Mutex locks in Daisy

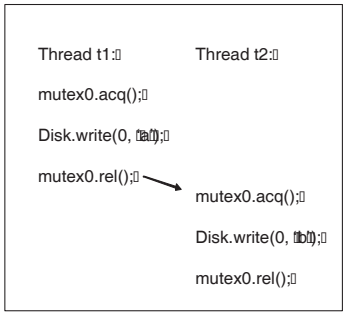

Fig. 14. Writes on a block

We applied our approach first on race condition detection since the property is well defined and also crucial for multi-threaded programs. As discussed previously, the tool needs to repeat the VC generation and property checking for every shared variable. To improve the analysis efficiency, the tool only checks one of the array elements instead of all of them. Figure 12 shows the results of the experiments. T_{pre} is the time for the pre-processing, T_{vc} is the time of the VC generation, and T_{φ} is the time used to detect the races. Note that T_{vc} and T_{φ} are the time cost for all checked shared variables instead of one, so there is a big difference for Daisy, raytracer and tomcat because of their large number of shared variables. But for a single variable, according to the experiments, even though the possibly worst cost of the VC generation is $O(|\xi|^2)$, it usually takes time similar to the pre-processing, which is linear to the number of events. This shows that the backtracking is rare in practice. The performance of the verification is also reasonable, which may be credited to the specific algorithm for detecting data races. Note that sometimes it takes less time to detect the races than to generate VCs because the tool returns right after it finds the bug.

Program	T_{pre}	T_{vc}	T_{φ}	Bugs	False
Banking	1s	2s	5s	1	0
Http-Server	0.2s	0.3s	0.3s	2	0
Daisy	5s	3m	30s	1	0
Daisy-2	29s	5m	30s	2	0
Raytracer	5m	30m	10m	1	0
Tomcat	2m	1.5h	3h	0	0

Fig. 12. Analysis results.

Our tool did not reveal any new races in these programs, but found almost all the known races in them without false alarms. For Tomcat, the lack of warnings may be due to the limited coverage of our testing, and may also imply that a more comprehensive instrumentation is needed. We will try to improve both after optimizations of the prototype are finished.

Below we provide a more detailed discussion using Daisy-2. Figure 13 shows the write functions on the disk and the implementation of Mutex locks in Daisy. The disk contains a byte array and provides functions similar to the RandomAccessFile in Java, e.g. *seek*() and *writeByte*(). The whole disk is logically divided into blocks, which are used to store files. Every block or file is protected by a specific Mutex lock to achieve better performance.

```
class Mutex {
...
    synchronized void acq() {
        if (locked) this.wait();
        locked = true;
    }
}
```

Fig. 15. Buggy lock acquiring

Although efforts have been made to assure the well-defined synchronization for the disk, there is a datarace reported. The datarace is caused by the usage of the *seek* function of the disk, which sets a pointer for the following file operation. Since the pointer is unique for the whole file, when two threads try to read/write different blocks, they will compete for the pointer without common locks. Noticing that every read/write on the disk is not protected by the system lock, the lock-set based algorithm will produce a large number of false alarms for disk accesses, which will overwhelm the real data race. In our approach, because the loop in the lock implementation is regarded as a non-terminating one, the following accesses to the disk have control-flow dependence on the loop. Therefore, accesses to the same disk block (a continuous area in the underlying array) are ordered by the read/write on the shared Mutex lock, as shown in Figure 14. So no races are reported for

accesses to the array. But when two threads try to access different blocks, they use different Mutex locks and a race on the file pointer will be reported.

Furthermore, let us consider the buggy implementation of Mutex in Figure 15. In this code, the while loop is replaced by a if statement, which will cause errors if multiple threads are waiting for the same lock. However, because of the causal partial order caused by read/write on the *locked* field, the potential datarace will not be detected by the traditional causal partial order based approaches. In our approach, because accesses to the shared memory, including the file pointer and the specific block in the disk, are out of the control scope of the if statement, there is no causal partial order between the two writes in Figure 14. Therefore races will be reported.

5.3 Verifying Safety Properties

Because of the time limitations, we leave further evaluation of verifying safety properties of more real applications using the prototype to our future work. In this section we use a simple example to illustrate the detection of potential safety property violations. Figure 16 shows a JAVA code fragment of an HTTP client taken from [20]. The client first requests access to the server. If not granted, it adds itself into a waiting queue (*/*1*/*) and then suspends itself (*/*2*/*), waiting for another client to resume it. If granted, it does its work and then resumes a waiting client, if there is any (*/*3*/*). The client continuously requests access to the server in a loop. To avoid dataraces, accesses to the waiting queue are synchronized.

```
public class HttpClient extends Thread {
 private static Vector suspendedClients = new Vector();
 ... unrelevant code ...
 public void run() {
  while (true) {
   ... request server access ...
   if (!accessGranted) {
/*1*/
    synchronized (suspendedClients) {
     suspendedClients.add(this);
    }
/*2*/
    suspend();
   } else {
    ... work with server ...
    synchronized (suspendedClients) {
     if (!suspendedClients.isEmpty()) {
/*3*/ ((HttpClient)suspendedClients.remove(0)).resume();
     } // end of if
    } // end of synchronized
   } // end of if
  } // end of while
 } // end of method
} // end of class
```

Fig. 16. Java code fragment of an Http client.

There are two subtle concurrency errors in this code. The first is as follows. Suppose that a client's access is denied and that, right before it adds itself to the *suspendedClients* queue (at */*1*/*), the thread scheduler delays it long enough that all the other clients terminate their jobs. Our client then continues to add itself to the waiting queue, but, unfortunately, there is no other client working with the server to ever resume it. Thus, the server ends up being idle while there is still a client waiting to be granted access unless another client requests and is granted access, eventually resuming the starved client. This error yields a violation of a liveness property, namely that "any suspended client will be eventually resumed", which, unfortunately, is *not monitorable* [13]

The other concurrency error is as follows. Suppose that a client is denied access, then puts itself into the waiting queue, and then right after releasing the lock but before suspending itself (at */*2*/*) is delayed long enough by the thread scheduler to allow another client to remove it from the waiting queue and resume it (*/*3*/*) – resume has no effect if the thread is not suspended. Then the thread regains control and continues to suspends itself. Now there is no information about its suspension in the waiting queue, so no other client will ever resume it: this client is suspended forever.

Both errors are not very likely to occur in a particular execution, so they are hard to catch by testing; and even if they occur during a testing session, they may be hard to locate. Moreover, since these are concurrency errors caused by non-deterministic thread interleaving, it is difficult to reproduce them. Hence, detection, localization and correction of these subtle bugs will be a time-consuming challenge even for experienced programmers.

By using our approach, one can detect the second bug in a single run. The bug can be detected as a datarace on the client if *suspend* and *resume* are considered as non-pure functions, as we did in Section 5.2. However, even with the reported datarace, it is still unclear if there is a real bug or not, because synchronizing *suspend* will lead to deadlocks. In fact, the error is essentially a violation of a safety property for using *suspend* and *resume*: for any thread, calls to *suspend* and *resume* alternate and start with a *suspend*. This can be

specified as a regular pattern, namely *(suspend resume)**, disregarding any other irrelevant events. Figure 17 shows the specification of this property in our approach.

```
Logic = ERE;
Event suspend: called(void suspend());
Event  resume: called(void  resume());
Formula: (suspend resume)*
```

Fig. 17. Specification for the usage of suspend/resume

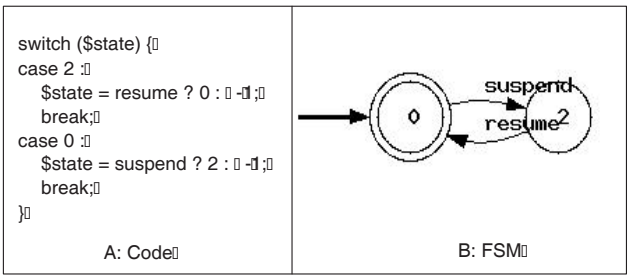

Fig. 18. Generated property monitor

The syntax of this specification is similar to the MOP specification [1]. Basically, the first line states that the underlying formalism is that of extended regular expressions (ERE). This way, the tool knows which logic plug-in to use for generating the monitoring code. Then the events to monitor are declared, which form the atoms over which the requirements are then formalized as a regular expression. The *called* event is qualified with the current object, not with the caller, and can also bind the arguments of the called method for further use in warning messages or recovery (not the case here). For this specification, the generated monitoring code is shown in Figure 18. Using this monitor, our tool is able to detect and locate the violation in less than a second.

6 Conclusion and Future Work

This paper introduces an runtime based approach to predicting potential safety violations from a successful run of the multi-threaded program. In this approach, dynamic information, i.e. the causal partial order among events and lock-atomicity, is combined with static information about the program, including the control scopes of statements and alias equivalence sets, to improve the completeness while still preserving the soundness of the analysis. This approach has been applied in several non-trivial multi-threaded Java programs and the results are encouraging.

The current prototype needs to be optimized to achieve better performance in practice. For example, for the race condition detection, we will implement a lock-set algorithm in the pre-processor to get rid of those well-protected variables immediately to avoid redundant analysis. Another optimization is to handle the array as one entity if necessary. More improvement can be achieved based on some static analysis, e.g., data slicing can be applied before hand to determine those variables related to the relevant events, minimizing the instrumentation of the program. This way, the performance of the instrumented program can be reduced and the analysis can be more efficient since fewer events are generated.

References

1. F. Chen, M. d'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *International Conference on Formal Engineering Methods (ICFEM)*, LNCS, pages 357–372. Springer, 2004.
2. F. Chen, M. d'Amorim, and G. Roşu. Checking and correcting behaviors of java programs at runtime with java-mop. In *Runtime Verification (RV)*, LNCS, pages 546 – 550. Springer, 2005.
3. J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.
4. Daisy website. http://research.microsoft.com/ qadeer/cav-issta.htm.
5. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 286. IEEE Computer Society, 2003.
6. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

7. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages(POPL)*, pages 256–267. ACM Press, 2004.
8. Website for java grande. http://www.javagrande.org/.
9. M. Kamkar. PhD thesis.
10. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication ACM*, 21(7):558–565, 1978.
11. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES and APPLICATIONS (OOPSLA)*, pages 105–106. ACM Press, 2000.
12. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 167–178. ACM Press, 2003.
13. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Journal of ASE (to appear)*, 2005.
14. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transaction of Computer System*, 15(4):391–411, 1997.
15. K. Sen, G. Roşu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 3535 of *LNCS*, pages 211–226. Springer, 2005.
16. K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer (STTT) (To Appear)*, 2005. previous version appeared in TACAS'04, LNCS v2988, 123-138.
17. K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, pages 337–346. ACM Press, 2003.
18. F. Tip. A survey of program slicing techniques. Technical report, 1994.
19. Tomcat. http://jakarta.apache.org/tomcat/.
20. S. Ur. Website for multi-threaded problems. http://qp.research.ibm.com /QuickPlace /concurrency_testing /Main.nsf.